MA 3046
Matrix Analysis
Introduction to the Laboratory Segment

Practical computation represents an essential, mandatory portion of this course. Matrix analysis and computation are not sterile, theoretical subjects. On the contrary, they lie at the core of virtually all modern engineering and design activities. Moreover, they reflect a discipline where the theory almost never lurks too far in the background - seriously erroneous results commonly occur whenever matrix numerical methods are employed in situations where they theoretically are "outside of their envelope."

These are intended to be laboratories in the classic sense that they often require you to carry (numerical) "experiments." These experiments, most of which are contained in short, usually already-written programs, have been specifically designed to reinforce or to expand on theoretical points which have been made in class. For the most part, you should work these laboratories like any other experimental laboratory, i.e. review the theory before hand, then conduct the experiment (i.e. run the programs), record the results, analyze the data, compare the observed behavior to theoretical predictions, and draw conclusions about the validity of the theory from the (numerical) behavior observed.

Some of these laboratories will be graded. Some will be just for your own learning. Some will serve as the starting point for some subsequent, graded project. You will be told at start of each laboratory class whether or not that particular one will be graded and, if so, when you will need to submit your report. Even when a particular laboratory is not graded, however, we strongly suggest that you not treat it too superficially. There is a very good chance that at least some of the concepts covered in that laboratory will reappear either in later graded projects or on in-class examinations.

Our policy in the course with respect to preparation of graded laboratory assignments is fairly liberal, in the sense that we generally encourage your freely discussing interesting or perplexing aspects of these laboratories with other students or with the instructor. Our primary expectation is that, when you finally submit any project for grading, the final written product is your own writing, that you acknowledge any extensive or critical assistance that you may have been given, and that you are reasonably able to explain or expand upon any statements you have made or conclusions you have reached.

Lastly, we strongly recommend you approach these laboratories with an open, and even slightly skeptical attitude. Expect and be watchful for occasional surprises! Numerical matrix analysis can involve some very subtle effects, which may be evident only in hindsight. That's one aspect that make it such an interesting field of study.

[This Page Intentionally Left Blank]

MA3232 - Numerical Analysis
The Workstation Environment

General Comments

These laboratories are designed to be run on the Windows 2000/NT workstations located in various laboratories located around campus. These workstations utilize the usual Microsoft Windows environment for communication (i.e. entering commands, editing files, etc.) and standard IEEE arithmetic for computation. Most of the programs and exercises in these laboratories can also be successfully (and often better) run on properly-configured Unix workstations, such as those found in some high-end engineering laboratories that still exist at various locations around the campus. However, you should be aware that non-Windows machines may produce slightly different results, depending on the particular machine and version of the relevant software installed there.

The prerequisites for this course include courses that should have been taught, at least in part, in one of the campus computer laboratories, and familiarity with the Matlab programming language. Therefore, these laboratories presume that you have already used and are somewhat familiar with NT, or similar Windows workstations, and that you also have minimal Matlab skills. (Although not a stated course prerequisite, some exposure to the Maple symbolic computation language may also be of benefit.) If you wish to briefly review Matlab, we suggest the following reference:

- *MATLAB Primer*, by Kermit Sigmon, Fourth Edition

Windows Skills and Commands

In general, besides familiarity with Matlab, you will need certain Windows skills in order to complete the laboratories. The most important of these skills are the abilities to:

- Open and close certain command windows
- Navigate between different (sub)directories
- Copy and move files
- Create, edit and save files

All of the above functions can, in general, be accomplished by clicking on the appropriate icon on the Windows Desktop, or by choosing properly from within some drop-down menu list. For example, an Internet browser window can be opened by mouse double-clicking on the appropriate icon (either Internet Explorer or Netscape) on the desktop. Alternatively, one may single click on the **Start** button on the left of the Windows Taskbar (at the bottom of the screen), then on **Programs**, then finally on the **Internet Explorer** icon within that list.

Unfortunately, the current Windows systems on campus are neither completely uniform across campus nor completely reliable, and the Matlab version found in the workstations in Glasgow may contain a somewhat different configuration than are available with any of the versions of Matlab found in the engineering laboratories. Therefore, we strongly recommend that you create, on your own personal desktop (strictly speaking as part of your *Roving Profile*), dedicated icons for the both the Glasgow LRC and the engineering departments versions of Matlab. You can do this by

(1) Locating the program icon for the desired program. (It should be either directly on the desktop or in a Desktop Folder called **Lab Specific Applications**

(2) Right mouse clicking on the desired program icon.

(3) Selecting **Copy**.

(4) Moving to any blank spot on the desktop, right-clicking the mouse, and selecting **Paste**.

(5) A copy of the icon should appear on the desktop, and will now follow you wherever you go on campus.

The programs which comprise the numerical "experiments" in these laboratories will generally be located on the NPS web, under either the Mathematics Department Web Site:

http://www.math.nps.navy.mil/∼art/ma3046

or the Intranet URL:

http://intra.nps.navy.mil/∼ma3046

and you should be able to be access, view and download them using any standard web browser on any *on-campus* system. (Because of security dictates, the Intranet site is not reachable from off-campus.) We strongly recommend that you bookmark these sites, and download from them into a subdirectory you have specifically created to hold **only** programs associated with this course. You can create subdirectories by double clicking on the **My Computer** icon, navigating into the parent directory where you want the new subdirectory located, then right-clicking on any blank space within the directory window; and finally selecting **New** and then **Folder** from the resulting drop down menu.

Lastly, the programs on these sites are updated as (usually minor) changes are made to them. Therefore, they may not *exactly* match the listings in this manual. However, rest assured you will be told when any significant changes are made to them!

MA 3046 - Matrix Analysis
Laboratory Number 1
Basic Features and Properties of MATLAB

The stated prerequisites for this course include familiarity with MATLAB in the NPS workstation environment, a familiarity which you should have gained in any number of other courses. (Actually, anyone with prior experience in any programming language such as BASIC, FORTRAN, C, etc., can, with a little effort, probably learn the basics of MATLAB on their own while taking this course.) In this laboratory, we shall briefly review some of the basics of MATLAB, the look at some aspects of MATLAB you may not have seen previously, and, along with these, give you an opportunity to become a bit more familiar with the specific workstations that we'll be using.

MATLAB (short for *Matrix Laboratory*) is a package designed for both practical computations and theoretical investigations of numerical methods and computation. As its name probably indicates, its original forte was matrix computations, however the current capabilities of MATLAB are far greater. MATLAB versions exist for mainframes (e.g. the IBM at NPS), Unix workstations, and PC's.

MATLAB is actually based on a complex suite of C programs that implement certain "primitive" subroutine operations, plus an interface driver routine which converts input, "English-like" and mathematical expressions into the correct subroutine calls. Thus, a MATLAB user wishing to compute the product of two matrices may simply type a statement like:

c = a∗b

and MATLAB will then (*completely transparently to the user*), interpret this as requesting a matrix multiplication, and issue a command, similar to, for example, the FORTRAN command:

call matmul(c,a,b,nrowa,ncola,nrowb,ncolb)

to actually multiply the correct matrices and store the resulting product appropriately.

On most PC systems, MATLAB is started by double-clicking on the appropriate icon. (On other, e.g. Unix systems, it may also be started by issuing the command **matlab** from within any command window.) After a "brief" delay, the MATLAB welcome and command prompt (">>") will appear. From here on, the user is free to create virtually any allowable algebraic, functional, or matrix operation or calculation by simply typing in the appropriate command(s). For example, you can obtain some appreciation of the power of MATLAB by issuing the **demo** command after the prompt appears.

Matrices, e.g.

$$\mathbf{a} = \begin{bmatrix} 1 & 3 & 1 \\ -1 & 2 & 4 \end{bmatrix}$$

can be created in MATLAB either by simply entering the elements, one at a time, e.g.:

$$\begin{aligned}\mathbf{a}(1,1) &= \ 1\ , \\ \mathbf{a}(1,2) &= \ 3\ ,\end{aligned}$$

etc. or by entering, on a single line,

a = [1 , 3 , 1 , ; -1 , 2 , 4]

where [indicates the start of the matrix, **;** the end of a row,] the end of the matrix, and the commas are optional. (Data or commands that are too long to fit on a single MATLAB command line may be continued onto the next line(s) by simply typing **...** , then hitting return, and just continuing to enter the data or command on the new line.)

MATLAB also creates and accesses vectors and submatrices using the **:** symbol to denote a *range* of values. For example, the statement

x = [0 : 10]

produces the same vector as would

x = [0 1 2 3 4 5 6 7 8 9 10]

while the statement

asub = a(1 : 2 , 2 : 3)

will produce the (sub)matrix consisting of the first and second rows and second and third columns of **a**, e.g. if we were to use this command with the matrix created above, we would obtain:

$$\mathbf{asub} = \begin{bmatrix} 3 & 1 \\ 2 & 4 \end{bmatrix}$$

(Note that since in this case we actually included all of the rows of **a**, we could also have just entered the command:

asub = a(: , 2 : 3)

The range feature allows us to operate with and exploit, wherever possible, the simplicity and power of *block matrices.* Block matrices are simply matrices, each of whose elements is itself a matrix. (While there are some minimal compatibility requirements that must be satisfied when performing algebraic operations using block matrices, in general one operates algebraically with them just as if they were "normal" matrices of numbers.)

Information about specific MATLAB commands may obtained by either selecting **Help** from the drop-down menu at the top of the window, or with the **help** command. Simply typing **help**, followed by the name of a specific command, e.g.

help chop

will generally give sufficient information about syntax that one can then use that command. Because the **help** command uses pure text, it is markedly faster than the menu if you know the name of the function you want more information about. If you have only a general idea about the category of problem you are working on, and are searching for possible programs to solve it, the menu is probably preferable.

Actually, very little of MATLAB is written in C. Most MATLAB commands and functions are really just text files of sequences of other MATLAB commands. These are commonly called **m**-*files*, and are generally clearly identified because their file name ends in **.m**, for example **chop.m**. (That most MATLAB commands and functions are written in the English-like MATLAB syntax greatly simplifies the inspection and analysis of MATLAB algorithms.) Users can also create their own commands in MATLAB, simply by using the built-in MATLAB text editor to create an appropriate (**ascii**) **m**-file of other commands. (You should never try to create MATLAB files with Microsoft Word, or any other general word processor!)

MATLAB actually allows two general types of user-written **m**-files, scripts and functions. The major difference between these is that functions, such as **chop.m**, are used with a set of input argument variables, and produce a specific output value (or set of values), e.g.

z = exp(-1.) produces **z = 0.3679** , and

while scripts allow a much wider variety of tasks to be executed. (Both types, however, still have the **.m** file name ending.) More information about scripts and functions can be obtained by giving the commands:

help function and **help script**

```
% This script demonstrates both the use of MATLAB scripts,
% and one of the effects of using floating-point arithmetic.
%
  a = 4/3
  b = a - 1
  c = b + b + b
  d = 1 - c
```

Figure 1.1 - Listing of Program **matrevpr1.m**

The program (**m**-file) **matrevpr1.m**, which is shown in Figure 1.1, represents one simple example of a MATLAB script. This script simply "automates" four sequential MATLAB commands and produces a rather interesting result!! (To fully appreciate this result, you should first work through these calculations by hand.)

By contrast, the **m**-file **ge_basic.m**, which is shown in Figure 1.2, is an example of a function **m**-file. This particular function implements "normal" Gaussian Elimination on a

```
  function [ uwork ] = ge_basic( aaug )
%
  [m,n] = size(aaug)
  uwork = aaug ;
%
  working_row = 1  ;
  working_col = working_row  ;
%
  while (working_row < m)*(working_col < n)
     i = working_row ;
     j = working_col ;
     if ( uwork(i,j) == 0 ) ; %%%%   check for zero pivot ;
         disp(' ');disp('  ');
        disp('*****   Error - zero in pivot position - current array is');
         uwork
         error('exiting. . . ');
     end
%
     for ii =  i+1:m
         lcoef          =  uwork(ii,j)/uwork(i,j)  ;
         uwork(ii,j)     =  0 ;
         for jj = (j+1):n
             uwork(ii,jj) = uwork(ii,jj) - lcoef*uwork(i,jj) ;
         end
     end
     working_row = working_row + 1;
     working_col = working_row    ;
   end
%
```

Figure 1.2 - Partial Listing of Program **ge_basic.m**

matrix (normally an augmented matrix), with no row interchanges, in the case when no zeros appear on the diagonal. (Note that, because of space considerations, the listing in the figure is a somewhat abbreviated version of the full program, which not only performs the computations, but outputs all of the intermediate results as well.) This **m**-file is clearly identified as a function by the first line, i.e.

function [uwork] = ge_basic(aaug)

which contains the keyword **function**, and goes on to indicate that, in this case, the input to this function is an array, and the output another array. (Specifically, the output is the upper triangular matrix that results from Gaussian elimination, without row interchanges, applied to **aaug**.) In addition, MATLAB also provides a number of fairly simple, yet powerful commands for producing high-quality standard, semi-logarithmic and logarithmic two and three-dimensional graphs. We shall investigate some of these capabilities in this laboratory.

As discussed in class, the Standard Inner Product in Complex Euclidean space ($\mathbb{C}^n$) can be defined, computationally, as:

$$\overline{\mathbf{u}}^H \mathbf{v} = \overline{u}_1 v_1 + \overline{u}_2 v_2 + \cdots + \overline{u}_n v_n$$

where $\mathbf{u}^H \equiv \overline{\mathbf{u}}^T$ is commonly called the *Hermitian* (or *adjoint*) of $\mathbf{u}$, although some texts uses the slightly more nondescript symbol $\mathbf{u}^*$. (Similarly, the Hermitian of a complex-valued matrix is defined by

$$\mathbf{U}^H \equiv \overline{\mathbf{U}}^T \quad \Longleftrightarrow \quad u_{ij}^H = \overline{u}_{ji} \ ,$$

or $\mathbf{U}^*$.) Note that the MATLAB operator $'$ produces the Hermitian, and so

$$\mathbf{u}^H \mathbf{v} \text{ in mathematical symbols} \quad \longleftrightarrow \quad \mathbf{u}' * \mathbf{v} \text{ in MATLAB}$$

(Note, however, that the Hermitian and the transpose are identical only when **all** the entries of the matrix or vector are purely real. Also note that, unfortunately, many texts define the Standard Inner Product in complex spaces in such a way as to results that are conjugates with those produced by our definition! This can produce no end of confusion, although no important theoretical results are affected. Lastly, also note that other, *non-standard* inner products can also be defined in either $\mathbb{C}^n$ or $\mathbb{R}^n$. However, again, we shall not persue those here.)

Having a computational definition of the inner product available in $\mathbb{C}^n$ also permits us to generalize concepts such as magnitude (length) and direction (angle) to higher-dimensional spaces, where ordinary geometrical constructs fail. For example, we define the "length" of a vector in $\mathbb{C}^n$ to be:

$$\| \mathbf{u} \| \equiv \sqrt{|u_1|^2 + |u_2|^2 + \cdots + |u_n|^2} = \sqrt{\mathbf{u}^H \mathbf{u}}$$

We can also extend the concept of mutually perpendicular vectors from $\mathbb{R}^2$ and $\mathbb{R}^3$ into $\mathbb{C}^n$ by calling a set of non-zero vectors $\boldsymbol{B} = \left\{ \mathbf{b}^{(1)}, \ \mathbf{b}^{(2)}, \ \ldots, \ \mathbf{b}^{(k)} \right\}$ *orthogonal* if:

$$\mathbf{b}^{(i)^T} \mathbf{b}^{(j)} = 0 \ , \quad i \neq j \tag{1}$$

Using orthogonal basis vectors to represent other vectors proves distinctly advantageous when we need to find coordinates, etc. Specifically, if the $\mathbf{b}^{(i)}$ form an orthogonal basis, and if a general vector $\mathbf{x}$ is expressed in terms of this basis, i.e. if

$$\mathbf{x} = \mathbf{B} \ [\mathbf{x}]_{\mathbf{B}} \tag{2}$$

where $\mathbf{B} = [\,\mathbf{b}^{(1)} \vdots \mathbf{b}^{(2)} \vdots \cdots \vdots \mathbf{b}^{(n)}]$, then the coordinates of $\mathbf{x}$ in terms of the orthogonal basis $\mathbf{B}$ are given by:

$$[\mathbf{x}]_{\mathbf{B}} = \left(\mathbf{B}^H\mathbf{B}\right)^{-1}\mathbf{B}^H\mathbf{x} \qquad \Longleftrightarrow \qquad ([\mathbf{x}]_{\mathbf{B}})_i = \frac{{\mathbf{b}^{(i)}}^H\mathbf{x}}{{\mathbf{b}^{(i)}}^H\mathbf{b}^{(i)}} = \frac{{\mathbf{b}^{(i)}}^H\mathbf{x}}{\|\,\mathbf{b}^{(i)}\,\|^2} \tag{3}$$

(These formulas are a direct consequence of the observation that the columns of $\mathbf{B}$ become the rows of $\mathbf{B}^H$. Therefore, if the columns of $\mathbf{B}$ are orthogonal vectors, then, it also follows directly from (1) that $\mathbf{B}^H\mathbf{B}$ is a diagonal matrix, whose diagonal entries are just the squares of the lengths of the corresponding columns of $\mathbf{B}$.) Lastly, note we shall also show later that computing coordinates using (3) requires approximately significantly less computational effort than solving (2) via Gaussian elimination. So orthogonal bases appear to offer significant practical computational advantages.

Even greater computational simplification results when the basis vectors are **orthonormal**, i.e. when the elements of $\boldsymbol{Q} = \left\{\mathbf{q}^{(1)},\ \mathbf{q}^{(2)},\ \ldots,\ \mathbf{q}^{(k)}\right\}$ are not only orthogonal, but also satisfy $\|\,\mathbf{q}^{(i)}\,\| = 1\ ,\quad i = 1, 2, \ldots, k$. In this case,

$$\mathbf{Q}^H\,\mathbf{Q} = \mathbf{I} \tag{4}$$

and therefore we immediately have that $[\mathbf{x}]_{\mathbf{Q}} = \mathbf{Q}^H\,\mathbf{x}$.

Complex-valued matrices that satisfy (4) are called *unitary*, while real matrices with the same property (i.e. $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$ are, somewhat confusingly, called *orthogonal* (vice orthonormal). Clearly, in either case, such matrices must have columns which are orthonormal vectors with respect to the corresponding Standard Inner Product.

Unitary matrices have several important properties. The foremost follows from the fact that if $\mathbf{Q}$ is square, then

$$\mathbf{Q}^T\mathbf{Q} = \mathbf{I} \quad \Longrightarrow \quad \mathbf{Q}^T = \mathbf{Q}^{-1}$$

and therefore inverting an orthogonal matrix is “free.” In addition, orthogonal matrices can be shown to be **length-preserving**, in that

$$\|\,\mathbf{Q}\,\mathbf{x}\,\| = \|\,\mathbf{x}\,\|$$

One almost-immediate consequence of this result is that

$$\|\,\mathbf{x}\,\| \ = \ \|\,[\mathbf{x}]_{\mathbf{Q}}\,\| = \sqrt{\alpha_1^2 + \alpha_2^2 + \cdots + \alpha_n^2}$$

regardless of what orthonormal basis is used. (This result is basically the same as the so-called Parseval’s lemma, a common result in Fourier analysis.) The also are **angle-preserving** in the sense that the angle between $\mathbf{x}$ and $\mathbf{y}$ will be identical to the angle between $\mathbf{Q}\,\mathbf{x}$ and $\mathbf{Q}\,\mathbf{y}$.

Actually, one slight notational correction is in order here. There are actually several, essentially equivalent ways, of measuring "length" in $\mathbb{C}^n$, and we commonly call any such measure a **norm**. Thus far in this discussion, we have actually been using the so-called Euclidean norm, which we should probably more accurately denote as

$$\| \mathbf{x} \|_2 \equiv \sqrt{\mathbf{x}^H \mathbf{x}}$$

The Euclidean norm is **not** the same as, although certainly a member of the same "family" as other norms we have referred to, i.e. the so-called infinity norm:

$$\| \mathbf{x} \|_\infty \equiv \max_i |x_i|$$

and the one norm:

$$\| \mathbf{x} \|_1 \equiv \sum_{i=1}^{n} |x_i| = |x_1| + |x_2| + \cdots + |x_n|$$

All three of these can be calculated in MATLAB, using variants of the **norm( )** command. Having said this, we will continue to use $\| \mathbf{x} \|$ to stand for either a generic norm or precisely one of the above norms, whenever which is the case should be clear from the context of the discussion.

Lastly, all vector norms **induce** corresponding matrix norms, according the relationship

$$\| \mathbf{A} \| = \max_{\mathbf{x}} \frac{\| \mathbf{A}\,\mathbf{x} \|}{\| \mathbf{x} \|} = \max_{\| \mathbf{x} \| = 1} \| \mathbf{A}\,\mathbf{x} \|$$

(Note that among other aspects, this relationship will allow us to fairly "cheaply" obtain at least order of magnitude estimates of $\| \mathbf{A} \|$.)

Lastly, we would observe that one primary area of concern in practical numerical linear algebra springs from the fact that most "real-world" problems involve matrices that tend to be very large, e.g. $n \sim 10^5$ or even larger. With matrices of this size, simple-looking algorithms from introductory linear algebra courses, e.g. Gaussian elimination, which theoretically solves

$$\mathbf{A}\,\mathbf{x} = \mathbf{b}$$

exactly in a finite number of steps, can require a truly daunting number of computations, even on a "fast machine." A reasonable understanding of the approximate number of computations required for any particular algorithm, and the associated execution time therefore becomes curcial in determining for what **size** matrices the algorithm will actually be useful!)

In numerical analysis, we commonly use the term *computational complexity* of an algorithm as a synonymn for a count of the number of floating-point multiplications, divisions, additions and subtractions, or so-called floating-point operations (flops) required by that algorithm. As we shall later see in class, hand-counting the number of flops required for

```
  data = [ ] ;
%
  for NMATRIX = [ 100, 250, 500, 750, 1000, 1250, 1500 ] ;
      a = randn(NMATRIX) ;          % generate random matrix
      b = randn(NMATRIX) ;          % generate another
      nitern = fix(3000/NMATRIX) + 1 ;    %determine iterations
      tic                           % set stopwatch
      for nit = 1:nitern
          b*a ;                     % compute
      end
      data = [ data ; NMATRIX, toc/nitern ] ;     % record results
      pause(5)
  end
%
  loglog( data(:,1),data(:,2),'*')
```

Figure 1.3 - Partial Listing of Program **time_mult.m**

a given algorithm generally involves fairly straightforward, albeit often highly laborious analysis. Editions of MATLAB prior to Version 6 greatly simplified such analyses by providing a built-in function, called **flops**, which would count the actual number of flops during the execution of any operation or program. Regrettably, starting with Version 6, this capability was sacrificed, partly in exchange for markedly faster execution times, and partly because, as we shall also later see, execution times on modern computing platforms depend strongly not only on the computational complexity, but also on numerous aspects of both the hardware architecture and the software.

While no longer counting flops, MATLAB and Windows do provide tools that support at least rudimentary analysis of algorithm efficiency. The most relevant MATLAB one are the **tic** and **toc** commands. The first of these, in effect, starts an internal "stopwatch." From that point on, until that stopwatch is restarted by a subsequent call to **tic**, each call to **toc** reads out the elapsed time. In many instances, will provide sufficient information to allow us to draw meaningfully compare (relative) algorithm efficiencies. However, there are several drawbacks to using **tic** and **toc**:

(i) The minimum time measured is on the order of 0.01 seconds - a glacial period by many computer standards. Therefore, we often have to average times over numerous repetations of an algorithm.

(ii) Individual timings often show significant random fluctuations, due to the effects of other tasks the computer might be running at the same time, network loads, etc. Credible results again require averaging the results of multiple runs.

Figure 1.3 shows an example using **tic** and **toc** to investigate computational complexity. The code determines and records in an array the time required to compute the product of two square matrices, i.e.

$$\mathbf{BA}$$

for random matrices of sizes from between 100×100 to 1500×1500. Note we use a variable number of iterations to ensure we calculate representative times for small(er) matrices, and call **tic**, effectively resetting the "stopwatch," only immediately before the actual computational loop, so that we measure only the cost of the multiplications. Then, immediately after each iteration loop we use a call to **toc** to record the average time for a single calculation. (The **pause(5)** command inserts a five second pause after each different matrix size. This pause, which happens after the **toc** command, does not affect the timing, but will be useful later.) Finally, note we choose to display our results using a log-log scale plot. We have a very compelling reason for this choice. Specifically, in any case where behavior is proportional to some power, i.e. where

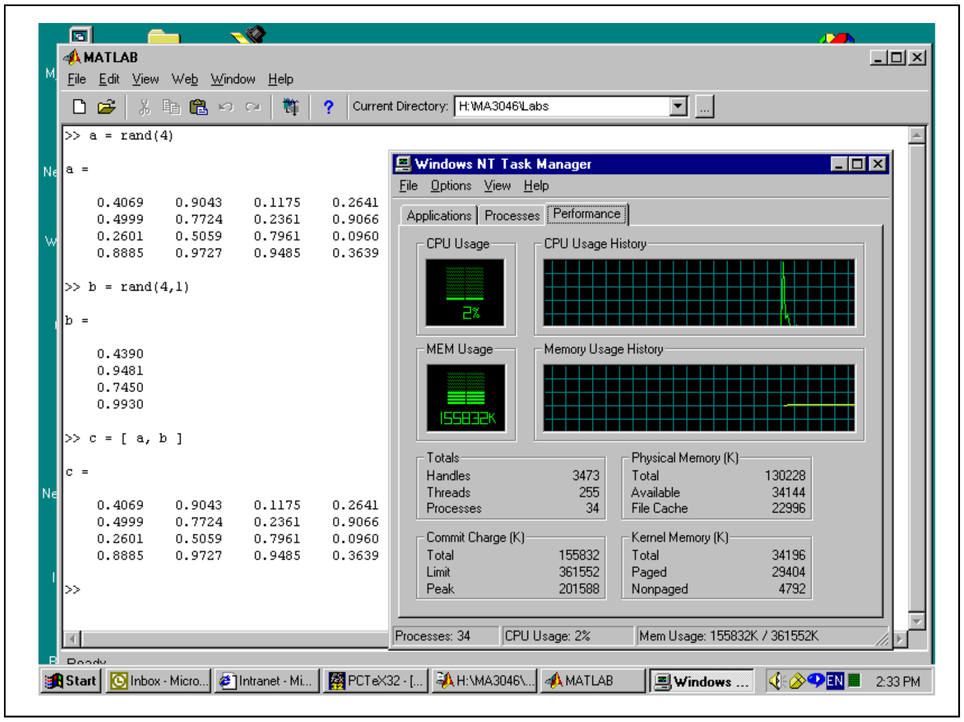
$$y \ = \ k\,x^{\alpha} \quad \Longrightarrow \quad \ln(y) \ = \ \ln(k) + \alpha \ln(x)$$

Therefore, on a log-log plot, this behavior will display as a *straight line* whose *slope* is α.

Finally, we additional insights into the efficiency of algorithms are available through the use of the Windows Task Manager. (The Task Manager can be launched by right-clicking your mouse on any *vacant* portion of the Windows Taskbar, then selecting Task Manager from the resulting pop-up menu.) The Task Manager actually contains three subwindows, *Applications*, *Processes*, and *Performance*. In this course, we shall use the Performance subwindow, a sample screen display from which is shown in Figure 1.4. Note that Task Manager shows two running graphs - one for current CPU utilization, the other for total current memory utilization. Both of these graphs are updated about once a second. In addition, this screen also display numerical values for other memory-related variables. Note, at this particular time, the CPU was running at only about two percent of capacity (i.e. essentially idle), and about 156K of memory was tied up by the various running processes. (So, obviously, were this measured in the middle of the execution of an algorithm, we would be observing and extremely inefficient process!) In this laboratory we shall observe Task Manager in conjuction with the **time_mult.m** program.

We would close by noting that, while MATLAB provides powerful capabilites for theoretical and computational numerical analysis, one drawback of MATLAB is that, by and large, it can only perform *numeric* computation, i.e. every MATLAB command causes a number of floating point computations to be made and the resulting number(s) displayed. Another area of computation that is beginning to come into its own is *symbolic* computation, which involves the manipulation of *algebraic expressions*, and not just numbers. The difference between these two types of computation is that, for example, a numeric computation package will only be able to tell you that

$$\int_0^1 xe^{-x}dx = 0.26424...$$



```
>> a = rand(4)

a =

    0.4069    0.9043    0.1175    0.2641
    0.4999    0.7724    0.2361    0.9066
    0.2601    0.5059    0.7961    0.0960
    0.8885    0.9727    0.9485    0.3639

>> b = rand(4,1)

b =

    0.4390
    0.9481
    0.7450
    0.9930

>> c = [ a, b ]

c =

    0.4069    0.9043    0.1175    0.2641
    0.4999    0.7724    0.2361    0.9066
    0.2601    0.5059    0.7961    0.0960
    0.8885    0.9727    0.9485    0.3639

>>
```

Totals	
Handles	3473
Threads	255
Processes	34

Physical Memory (K)	
Total	130228
Available	34144
File Cache	22996

Commit Charge (K)	
Total	155832
Limit	361552
Peak	201588

Kernel Memory (K)	
Total	34196
Paged	29404
Nonpaged	4792

Processes: 34 CPU Usage: 2% Mem Usage: 155832K / 361552K

Figure 1.4 - The Windows Task Manager

while a *symbolic* computation program will **also** tell you that

$$\int_0^1 xe^{-x}dx = 1 - 2e^{-1} \qquad .$$

As you may have already seen in your calculus (or other) courses, symbolic computation can greatly speed certain analyses, by allowing us to skip over lengthy, time-consuming but mechanical manipulations, and concentrate on analyzing the results of those manipulations instead. Therefore, although we shall not use it explicitly in this course, we would be remiss if we did not at least mention, in this initial laboratory, the symbolic computation language supported at NPS, a package called MAPLE. MAPLE provides numerous algebraic manipulation capabilities, including some symbolic matrix manipulation ones, as well as excellent graphics and some numerical computation and evaluation. As noted above, however, MAPLE's forte is symbolic manipulation, and, in general, MATLAB (or even FORTRAN or C) should be preferred for pure "number crunching."

Name: ______________________________

MA 3046 - Matrix Analysis
Laboratory Number 1
Basic Features and Properties of MATLAB

1. Login to your workstation and start MATLAB.

2. Use the **help** command to determine the proper use and meaning of the command or statements named **inv** and **eye**. Then create the matrix

$$\mathbf{a} = \begin{bmatrix} 1 & 3 \\ -1 & 2 \end{bmatrix}$$

and give the commands:

a. **inv(a)**

b. **inv(a)*a**

c. **eye(2)** - **inv(a)*a**

Does these result seem reasonable? What, if anything, can you infer about the working precision of MATLAB from calculation c.?

3. Use the **help** command to look at the **randn** command. Then generate a random 6×3 matrix

$$\mathbf{a} = \begin{bmatrix} \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{bmatrix}$$

Do the results appear to agree with the description provided by **help**?

4. Give the command

a(:)

and record the first ten entries:

What do the results appear to say about how MATLAB stores matrices?

5. Using the **randn** command, generate a random column vector ($\mathbf{x}$) with three elements.

$$\mathbf{x} = \begin{bmatrix} \\ \\ \\ \\ \end{bmatrix}$$

Then, using the random matrix generated in problem 3, give the commands

$$\mathbf{a * x}$$

and

$$\mathbf{x(1) * a(:,1)\ +\ x(2) * a(:,2)\ +\ x(3) * a(:,3)}$$

Does your result confirm that any matrix vector product is simply a linear combination of the columns of the matrix?

$$\mathbf{a * x} = \begin{bmatrix} \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{bmatrix}$$

6. Use MATLAB's range capability to partition the matrix generated in problem 3 into a 1×2 block format, i.e.

$$[\, \mathbf{A}_{11} \quad \mathbf{A}_{12} \,]$$

where $\mathbf{A}_{11}$ has two columns, and $\mathbf{A}_{12}$ has one column. Correspondingly partition the column vector $\mathbf{x}$ from problem 5 and then verify computationally that

$$\mathbf{A}\mathbf{x} \equiv [\, \mathbf{A}_{11} \quad \mathbf{A}_{12} \,] \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = [\, \mathbf{A}_{11}\mathbf{x}_1 + \mathbf{A}_{12}\mathbf{x}_2 \,]$$

Result:

$$\mathbf{a} * \mathbf{x} = \begin{bmatrix} \quad \\ \quad \\ \quad \\ \quad \end{bmatrix}$$

7. Give the command **help which** and study the response. Then give the following commands, and record the responses:

a. **which inv** Answer – ____________________

b. **which sin** Answer – ____________________

a. **which asec** Answer – ____________________

a. **which chop** Answer – ____________________

8. Give the command **help chop** and examine the output. Next, give, in sequence, the commands

```
chop( 27.321592 , 5 )

chop( 27.321592 , 4 )

chop( 27.321592 , 3 )
```

Are the answers what you expected?

9. Give the command **help script** and examine the output. Next, using a web browser pointed to the Laboratories link off

http://www.math.nps.navy.mil/∼art/ma3046 ,

copy the MATLAB script **matrevpr1.m** to your disk. Then, open a texteditor window (we recommend the built-in MATLAB editor) and examine this script. What do you expect the result will be?

10. Now run the script by entering the command **matrevpr1** in the MATLAB command window, or, if you're in the built-in MATLAB editor by simply hitting the F5 key. How well does the result support your earlier conclusion about the working precision of MATLAB?

(Note that this script, like all MATLAB scripts, does not normally show you, on the screen, the commands it is executing as it runs. If you want to see these, you have to give the command **echo on** before running the script. To stop further display of script commands, enter **echo off**.

11. Now, *from the command prompt*, give each of the commands in **matrevpr1.m** individually. Is the result what you expected??

12. Again using your web browser, copy the MATLAB script **ge_basic.m** to your disk, open a texteditor window, and examine this script.

13. Using MATLAB, created the augmented matrix **aaug** for the system studied on pages 5-9 of the text:

$$\begin{array}{rcrcrcr} 2x_1 & + & x_2 & + & x_3 & = & 1 \\ 6x_1 & + & 2x_2 & + & x_3 & = & -1 \\ -2x_1 & + & 2x_2 & + & x_3 & = & 7 \end{array}$$

Then, from the MATLAB command window, give the command

uwork = ge_basic(aaug)

Observe the steps in the elimination and compare them to the results you expect if you work the problem by hand. Also record the final result:

$$\textbf{uwork} = \left[\begin{array}{ccc} \quad & \quad & \quad \\ & & \\ & & \end{array} \right]$$

Are the results what you expected?

14. Create a vector **x** that divides the interval $0 \leq x \leq 10$ into values evenly spaced 0.05 apart by giving the command

x = 0 : 0.05 : 10

(Note this shows another variant of the range specification capability.) Then give the command

y = 4*x.∧3

What is this doing? (Note that the period here is *vital.*)

15. Now give the commands

plot(x , y) and **loglog(x , y)**

This will give you some idea of the power and simplicity of MATLAB's graphics.

16. Next create the matrix:

$$\mathbf{b} = \begin{bmatrix} 2i & (2+i) \\ 1 & (1-i) \end{bmatrix}$$

and then give the command:

$$\mathbf{b}' = \begin{bmatrix} \qquad\qquad\qquad\qquad \\ \\ \\ \\ \end{bmatrix}$$

Briefly describe how this does or not agree with what you expected?

17. Create the matrix

$$\mathbf{B} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & -1 \\ 1 & 1 & -1 \\ 1 & -1 & 1 \end{bmatrix}$$

and verify that the columns of **B** are orthogonal, but not orthonormal, vectors.

18. Normalize (convert to length one) each of the columns of the matrix created in part 17, and call this matrix

$$\mathbf{Q} = \begin{bmatrix} \qquad\qquad\qquad\qquad \\ \\ \\ \\ \\ \end{bmatrix}$$

19. For the matrix **Q** created in part 18, verify that

$$\mathbf{Q}^H\,\mathbf{Q} = \mathbf{I} \quad \text{but} \quad \mathbf{Q}\,\mathbf{Q}^H \neq \mathbf{I}$$

and therefore $\mathbf{Q} \neq \mathbf{Q}^{-1}$. Why is this not a contradiction?

20. Add the column

$$\begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix}$$

to the matrix $\mathbf{B}$ created in part 17,

$$\mathbf{B} \; = \begin{bmatrix} \quad & \quad \\ \quad & \quad \end{bmatrix}$$

and then normalize each column of that matrix

$$\mathbf{Q} \; = \begin{bmatrix} \quad & \quad \\ \quad & \quad \end{bmatrix}$$

21. For the new matrix $\mathbf{Q}$ created in part 20, verify now that both

$$\mathbf{Q}^H \, \mathbf{Q} = \mathbf{I} \quad \text{and} \quad \mathbf{Q} \, \mathbf{Q}^H = \mathbf{I}$$

Why should this now be the case?

22. Generate a random 4×1 vector

$$\mathbf{x} = \begin{bmatrix} \quad\quad\quad\quad \\ \\ \\ \\ \end{bmatrix}$$

23. Give the MATLAB **help** command to review the **slash** and **norm** functions.

24. For the vector $\mathbf{x}$ created in part 22 above and the matrix $\mathbf{Q}$ computed in part 20, also compute

$$\mathbf{aone} = \mathbf{Q}^H\,\mathbf{x} = \begin{bmatrix} \quad\quad\quad\quad \\ \\ \\ \\ \end{bmatrix}$$

and

$$\mathbf{atwo} = \mathbf{Q} \backslash \mathbf{x} = \begin{bmatrix} \quad\quad\quad\quad \\ \\ \\ \\ \end{bmatrix}$$

Why is this result either what you did, or did not expect?

25. For the vector $\mathbf{x}$ created in part 22 above and the matrix $\mathbf{Q}$ from part 20, also give the MATLAB commands

$$\mathbf{norm}(\mathbf{x})$$

$$\mathbf{norm}(\mathbf{Q} * \mathbf{x})$$

Why is the result either what you did, or did not expect?

26. Generate a random 5×3 matrix

$$\mathbf{a} = \begin{bmatrix} \quad & \quad & \quad \\ & & \\ & & \\ & & \\ & & \end{bmatrix}$$

and verify that its columns are **not** orthogonal.

27. Using the MATLAB **help** function, study the structure and use of the **rank( )** command. Then give the command

rank(a) Answer – ____________________

28. Next, create the matrix

$$\mathbf{b} = \begin{bmatrix} 1 & 3 & 2 & -1 \\ -2 & -1 & 3 & 1 \\ 1 & 2 & 1 & -2 \end{bmatrix}$$

then, using the matrix **a** created in part 26 above, compute

$$\mathbf{ahat} = \mathbf{a} * \mathbf{b} = \begin{bmatrix} \quad \\ \quad \\ \quad \end{bmatrix}$$

and finally give the MATLAB command

rank(ahat) Answer – ____________________

Why is the result either what you did, or did not expect?

29. Copy to your local directory the files **time_mult.m**. Then, open the MATLAB texteditor and *briefly* study it. Be sure before continuing that you're convinced it reasonably accurately calculates only the time(s) required to compute $\mathbf{b} * \mathbf{a}$. Once you're convinced it correctly does this, run it, observe the resulting graph and also print out the **data** array:

NMATRIX	**time**

Why does the graph here appear to be almost a straight line?

30. On the graph as produced in part 29, draw (by hand), the straight line which "best" fits the data. Then, based on the axes for that graph, determine the slope of this line.

Answer – ____________________

Briefly explain why your answer here either is or is not reasonable?

31. Using the fact that when both **a** and **b** are $n \times n$, computing $\mathbf{b} * \mathbf{a}$ requires approximately $2n^3$ flops, and based on your results from part 29 above, determine the effective computational speed of your PC:

Answer – ____________________ Mflops/sec

(where 1Mflop = one million floating point operations.) Compare this result to you machine's advertised CPU speed.

32. Finally, start the Windows Task Manager, and select the "Performance" window. Then rerun program **time_mult.m** as above, this time observing the performance display. Note specifically how the CPU utilization drops during the **pause(5)** commands, and how the memory used increases as the size of the matrices increases.